Süsteemprogrammeerimine keeles C

milles kõigepealt likvideerime võla, siis räägime taas pointeritest ja massiividest, massiividest funktsiooni argumentidena, dünaamilisest mäluhaldusest, funktsioonile teise funktsiooni argumendina andmisest, kahemõõtmelistest massiividest ja void pointeri kasutamise eesmärkidest.

Eelmises osas (Funktsiooniargumendid)

- Funktsioonile argumentide andmisel tehakse neist koopiad, millest lähtuvalt ei saa funktsioon oma argumente muuta (muudad koopiat)
- Kui funktsioon peab oma argumente muutma, antakse argument pointeriga. St funktsioonile antakse aadress, kust ta vajalikud muutujad leiab.

Eelmises osas (stringid)

- String on massiiv chare:

```
char jabur[]= {'j', 'a', 'b', 'u', 'r', '\0'};
char mittejabur[] = "mittejabur";
```

- Stringi lõpetab terminaator: '\0'

Eelmises osas (main())

- Main saab kaks argumenti:
  - integer: argumentide arv
  - stringide massiiv: argumendid ise
- Esimene argument on alati käivitatava programmi nimi
- ```
  int main(int argc, char **argv) { ... }
  int main(int argc, char * argv[]) { ... }
  ```
  ```

Eelmises osas (Stream)

- Failist lugemiseks või sinna kirjutamiseks tuleb avada *stream*.
- Operatsioonisüsteem annab programmile 3 standardset streami: stdin, stdout, stderr.
  - *FILE tüüpi ja vastava nimega muutujates
- Failide kasutamiseks saab neid teha ka ise:
  - FILE *fopen(char *name, char * mode);
  - vea korral tagastub NULL
- fprintf(), fscanf(), getc(), putc(), fgets(), fputs()
- fclose(FILE *filepointer);

Eelmises osas(Stream(2))

- Standardseid *streame* on võimalik suunata ümber

  Sisendi ümbersuunamine:

  ```
  minuproge < sisend
  ```

  Väljundi ümbersuunamine

  ```
  minuproge > väljund
  ```

  Sisendi ja väljundi ümbersuunamine

  ```
  minuproge < sisend > väljund
  ```

  `stderr` ümbersuunamine

  ```
  minuproge 2> error.log
  ```

Eelmises osas (Librad)

- Standardse *library* funktsioonide kasutamiseks tuleb vastava libra päis panna kompileeritavasse faili:

  #include <libranimi.h>

  - Matemaatika `#include <math.h>`
  - Stringid `#include <string.h>`
  - Sisend/Väljund `#include <stdio.h>`
  - Dünaamiline mäluhaldus `#include <stdlib.h>`

- Librad leiate enamasti kataloogist /usr/include

Eelmises osas (? : ja ++ ning --)

- Tingimuslik väärtustamine (**?** **:**):
  - *tingimus* **?** *väärtus_kui_tõene* **:** *väärtus_kui_väär*
- Suurendus ja vähendusoperaatorid (++, --):
  - Võimalik kasutada nii prefiksina (++i) , kui sufiksina (i++)

    i++ : kõigepealt väärtustub, siis suurendab muutujat

    ++i: kõigepealt suurendab muutujat, siis väärtustub

    - kui eraldiseisev, käitub identselt.

Täna kavas

- Makrod
- Header'id
- Pointerid ja massiivid
- Pointerid funktsiooniargumentidena
- Dünaamiline mäluhaldus
- Pointerid funktsioonidele
- Kahemõõtmelised massiivid
- Void pointer

Makrod

- C preprotsessori konstandile on võimalik anda parameeter, muutes ta makroks:

  #define *nimi*(*parameetrid*) *märgijada*

  NB: '=' märki ei ole.

- Makrod lahendatakse preprotsessoris enne kompileerimist

```
#define MAX_STR_LEN 20
#define IS_LOWER(c) ((c)>=‘a’ && (c)<=‘z’)
#define TO_LOWER(c) (IS_LOWER(c)?((c)–‘a’+‘A’):(c))
char arr[MAX_STR_LEN+1], *str;
….
for (str=arr;*str != ‘\0’; str++) {
  *str = TO_LOWER(*str);
}
```

Makrode näite selgitus

Eelmises näites:

```
char arr[MAX_STR_LEN+1], *str;
```

Teisendub kui:

```
char arr[20+1], *str;
```

Ning

```
*str = TO_LOWER(*str);
```

Esimeses järgus:

```
*str = (IS_LOWER(*str)?((*str)-'a'+'A'):(*str));
```

Teises järgus:

```
*str=(((*str)>='a'&&(*str)<='z')?((*str)-'a'+'A'):(*str));
```

Alles seejärel algab kompileerimine.

is*xxxx* makrod

- #include <ctype.h>
- int is*xxxxxx*(int c); //deklaratsioon
- Tegelikult on makrod
- isdigit(), isxdigit(), islower(), isupper(), isalpha(), isalnum(), isspace(), iscntrl(), ispunct(), isprint(), isgraph(), isascii()

Probleemid makrodega

- Probleemide põhjus on reeglina see, et makrod pole oma olemuselt C, vaid preprotsessori käsud.
- Näiteks: #define SQR(X) X*X
- Tehtejärjekorra viga:
  - SQR(a+b)
  - teisendub kui: a+b*a+b ja mitte (a+b)*(a+b)
- Lahenduseks saab tehete ümber ohtralt sulge lisada: #define SQR(X) ((X)*(X))

Lisaprobleemid

- Kõrvalefektid:
  - SQR(i++)
  - teisendub kui i++*i++, mis suurendab i-d 2 korda.
- Mõttetud funktsiooniväljakutsed
  - SQR(mingi_funktsioon(a,b,c))
  - funktsioon käivitatakse kaks korda
- Neile probleemidele ei ole otsest lahendust, seega tuleb ise tähelepanelik olla.

Funktsioon versus makro

Funktsioon:

- Argumente ei muudeta
- Omab määratud tüüpi
- Hoiab kokku käivitatavat koodi
- Võib anda teistele funktsioonidele argumendiks
- Funktsiooni väljakutse *overhead*

Makro

- Kõrvalefektid argumentidele
- Töötab (enamasti) erinevat tüüpi argumentidega
- Koodi ei dubleerita
- Ei saa argumendina ette anda
- *Overhead* puudub

Millal kasutada makrot

- Üldreeglid:
  - tehtav operatsioon on lühike, lihtne ja seda kasutatakse mitmes kohas (failis)
  - tehtav operatsioon on lühike, lihtne ja tihtikasutatav
  - operatsiooni on vaja teha erinevate tüüpidega

```
#define MAX(a,b) (((a)>(b)) ? (a) : (b))
#define SWAP(type,a,b) {type t=a; (a)=(b); (b)=t;}
```

Enumeration (nummerdustüüp)

- Tüübid, mis koosnevad vaid mingitest väärtustest, millel on sümboolsed nimed
- enum-i definitsioonid:

```
enum bool {FALSE,TRUE};  /* FALSE=0, TRUE=1 */
enum month {JAN=1,FEB=2,…,DEC=12};
enum colors {WHITE=1,BLACK,GREEN=8,RED};
```

- kasutamine:

```
enum bool b[10];
enum cond test = FALSE;
```

- Enum versus #define:
  - kompileerija kontrollib, debugger asendab tagasi

enum-i näited

```
enum day {sun,mon,tue,wed,thu,fri,sat};
  enum day d1;
  d1=fri;
enum {fir, pine } tree; // it is ok
enum tree {fir,pine} tree; //legal but not recommended
```

Switch

```
switch (month) {
case JAN: /* stmt */
…
case DEC:
   printf(“31 days\n”);
   break;
case APR:
…
case NOV:
   printf(“30 days\n”);
   break;
…
```

```
…
case FEB:
   if (leap_year)
       printf(“29 days\n”);
   else
       printf(“28 days\n”);
   break;
default:
   printf(“month error\n”);
   break;
}
```

break

- Katkestab tsükli täitmise enneagselt
- Kasutatakse for, while, do-while tsüklites
- Näide:

```
/* tagastab 1 kui "a" on kasvav, 0 muidu */
int monotonic(int a[], int N)
{
int i;
for (i=0; i<=N-1;i++) {
  if (a[i+1] < a[i])
  break;
}
if (i==N) return 1;
  else return 0;
}
```

continue

- Continue alustab kohe järgmise tsükli täitmist
- Näide:

```
char s1[12] = “string1”;
char s2[12] = “string2”;
char m[12];
int i,count = 0;
for (i=0;i<=12;i++) {
  if (s1[i] != s2[i]) continue;
  m[count++] = s1[i];
}
```

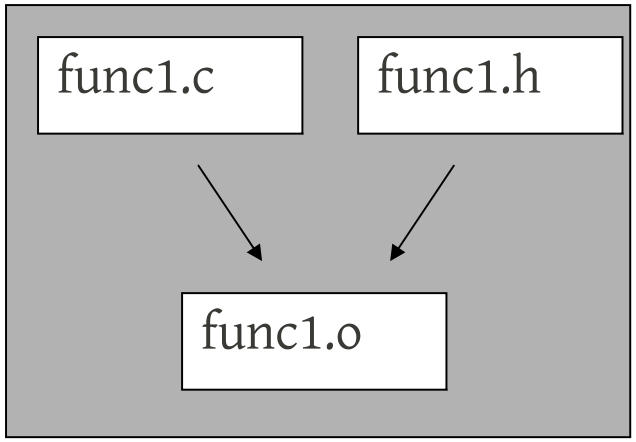
C Preprotsessor

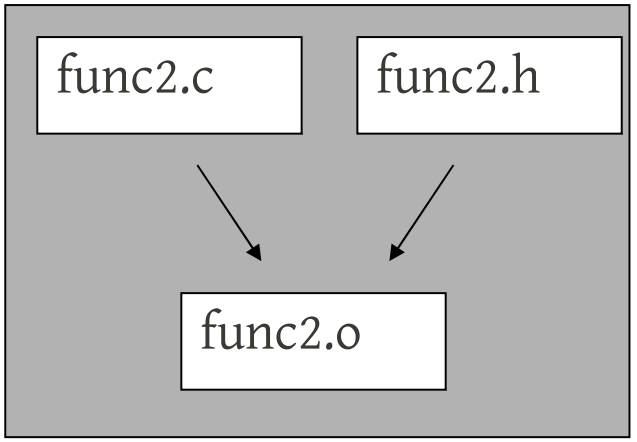
- Programm lastakse enne kompileerimist preprotsessorist läbi
- Preprotsessori direktiivid:
  - #include – lisab *header* faili (või suvalise faili)
  - #define – defineerib makro või konstandi
  - #ifndef, #ifdef, #endif – tingimuslik lisamine
  - #if, #else, #elif, #endif
  - #undef – tühistab definitsiooni

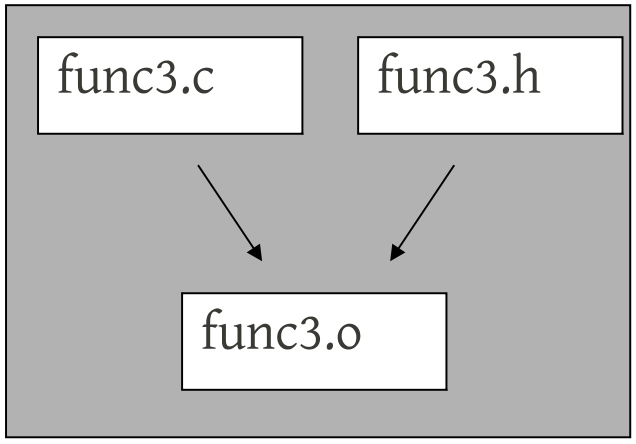
Header failid

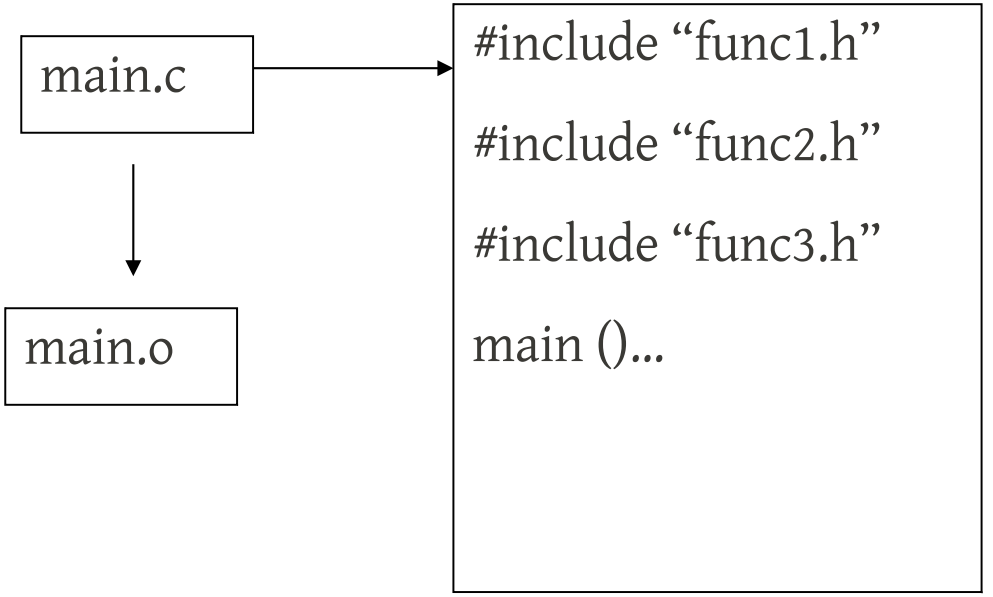
- Milleks on header failid?
- Lihtne viis kasutada varem defineeritud funktsioone (sisaldavad ainult prototüüpi)
- Modulaarsus: võimaldab kirjutada väikseid komponente ja neid ühendada
- Iga komponent omab **.c** ja **.h** faili
  - .c failis on funktsioonide definitsioonid
  - .h failis on funktsioonide prototüübid, konstandid, makrode definitsioonid
- Lihtsam siluda ja kasutada

Programmi struktuur

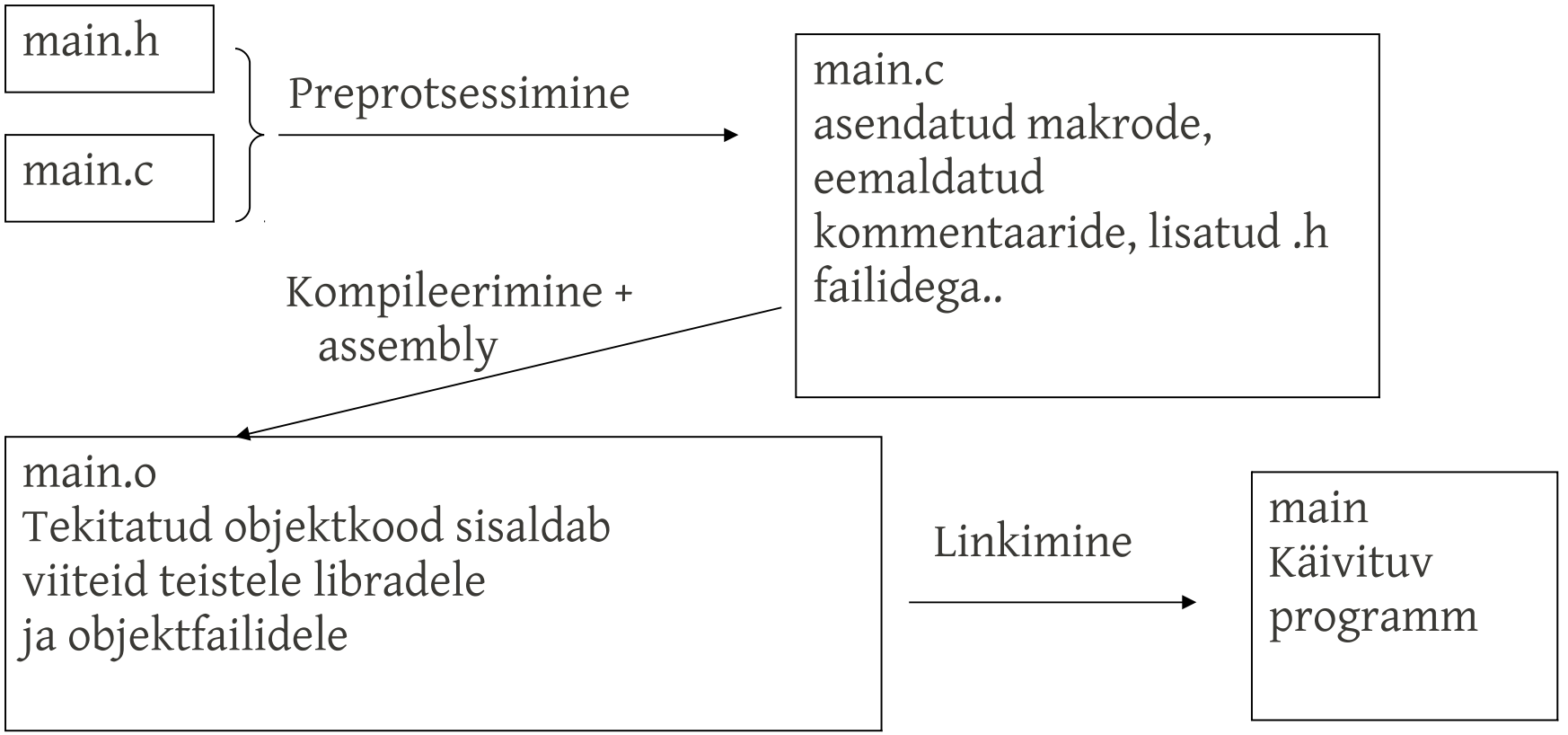








Programmi ehitamine



Tingimuslik teksti lisamine

- *Header* failid kasutavad tihti #ifndef käsku
- *Header*it mitu korda lisada on enamasti väga halb (pl.c lisab h2.h ja h1.h , mis samuti lisab h2.h)
- Selle vältimiseks tehakse *headeris* järgmine trikk:

```
#ifndef _header_name
#define _header_name

/* teiskordsel saabumisel seda siin ei täideta
  */
…

#endif
```

Mitmefaililise proge näide

main.c

```
#include "func1.h"
#include <stdio.h>
main()
{
   printf ("square of %d is
   %d\n", 2, sqr(2));
}
```

func1.c

```
#include "func1.h"
int sqr(int x)
{
    return x * x;
}
```

func1.h

```
#ifndef _FUNC1_H_
#define _FUNC1_H_
int sqr(int x);
#endif // _FUNC1_H_
```

Aadressid ja pointerid (1)

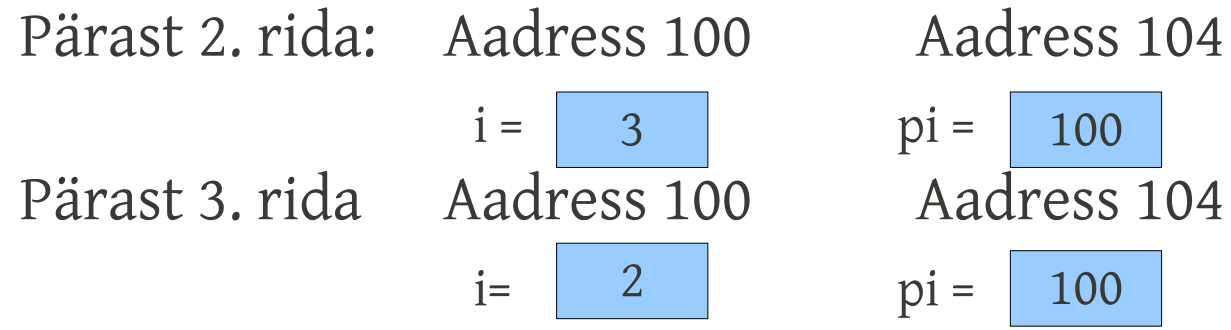
- Kõik arvutis olevad objektid paiknevad mingil aadressil
- Mõndadele C programmi objektidele saab nende mälus paiknemise aadressi järgi viidata
  - *Expression*&var saab väärtuseks var-i aadressi
- Ajutist mäluaadressi omavad objektid (numbrikonstandid, liittehted) pole & operaatoriga kasutatavad.
- Aadresse saab hoida pointerites

Aadressid ja pointerid (2)

- Kui pvar on pointer, milles on aadress, võtab * operaator (*dereference, indirection*) sellel aadressil paikneva sisu (tehe: *pvar).
- * operaatorit kasutame ka pointerite defineerimisel

```
1:  int i, *pi;          // *pi - pointer integerile

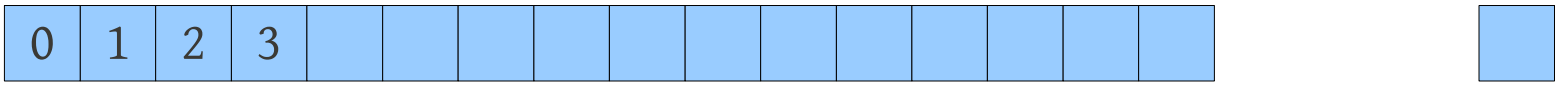
2:  i = 3; pi = &i;      // now (*pi == 3)
3:  *pi = 2;             // now (i == 2)
```



Aadressid ja pointerid (3)

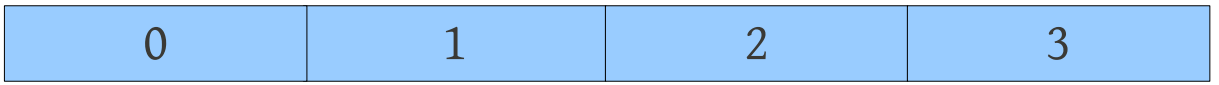
- Pointeris olevale aadressile * operaatori rakendamisel peame teadma andmete tüüpi
  - Seda seepärast, et erinevad andmetüübid võtavad erineva hulga mälu (char, int, float, double).

```
char c[N]; char *pc = &c[0]; // *(pc+1) == c[1];
```

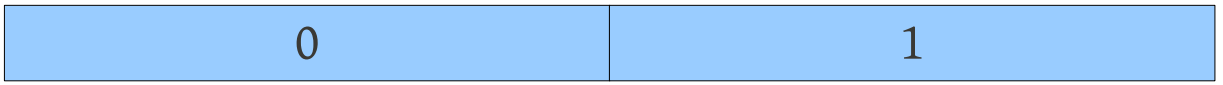


- 1 bait

```
int i[N]; int *pi = &i[0]; // *(pi+1) == i[1];
```




```
double d[N]; char *pd = &d[0];// *(pd+1) == d[1];
```



Pointerite võrdlemine

- Pointerite võrdlemine ei ole reeglina mõttekas tegevus
- Erand, mis reeglit kinnitab:
  - Sama massiivi elemendid paiknevad mälus järjest ja nende võrdlemine võib olla mõttekas
- Teine erand, mis reeglit kinnitab:
  - Võid alati kontrollida kas mingi pointer on NULL

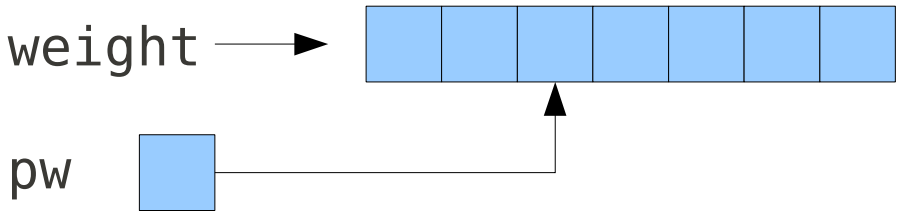
Pointer ja massiiv

- Kui on olemas definitsioon: **`double weight[LEN], *pw;`** kehtib järgnev.
- **`weight[i]`** on *expression*, mis viitab väärtusele, mis on salvestatud massiivi i-nda elemendina
- **`weight`** on pointer tüüpi *expression*, mis viitab massivi esimesele elemendile
  - Järeldus: (weight == &weight[0]) on **alati** tõene.
- C kompilaator teeb **`weight[i]`** korral sisemise teisenduse, mille tulemuseks on **`*(weight + i)`**.
- Pärast omistust **`pw = weight`** on **`pw[2]`** sama väärtusega, mis **`weight[2]`**
- **`pw`** ja **`weight`** põhierinevus on see, et **`weight`** on konstant ja seda ei saa muuta, samas kui **`pw`**-d võib muuta

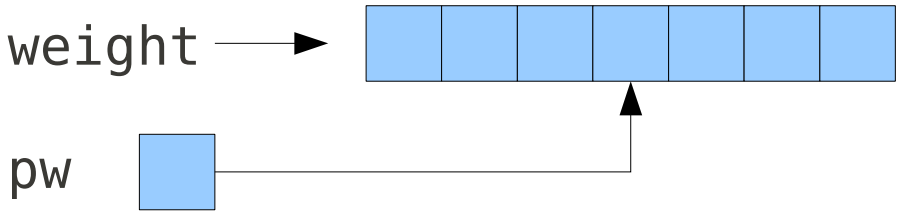
Pointer ja massiiv (Näited)

- Näide:

```
pw = weight + 2
```




```
pw++; // pw[0] on weight[3]
```



- Samas:

```
weight++;  /* VIGA! */
```

Pointerid ja funktsiooni argumendid

- C keeles saab argumente ainult koopiana anda (*pass-by-value*)
- See tähendab, et funktsioonile antud muutuja funktsiooni töö ajal ei muutu.
- Pointeritega saab sellest mööda hiilida:

```
Example:
void Swap(int *a, int *b)
{
  int t = *a; *a = *b; *b = t;
}
int i = 3, j = 4;
Swap(&i, &j);
```

- ülaltoodu on väljakutse viitamisega (*call by reference*)

Massiiv funktsiooniargumendina

- Massiiv on erand koopia reeglist.
- Kui funktsiooni argumendiks on massiiv, ei kopeerita kogu massiivi, vaid edastatakse aadress.
  - St sisuliselt ei ole tegemist erandi, vaid faktiga, et kopeeritakse massiivi pointer
  - int func vec[SIZE] puhul on func(vec) ja func(&vec[0]) samaväärsed;
  - definitsioonis on func(int *arr) ja func(int arr[]) identsed
  - massiivi osa edastamisel on samaväärsed näiteks func(vec+2) ja func(&vec[2])

Topeltviitamise meenutus

- Võtame

```
int a;
int *pa;
int **ppa;
```

- Mis on &a tüüp?
  - Pointer täisarvule (int *pa)
- Mis on &pa tüüp?
  - Pointer täisarvu pointerile (int *pa)
- Kas pärast omistust pa = &a on korrektsed?

```
*ppa = pa;
*ppa = &a;
int **ppa = &&a;
```

Dünaamiline mäluhaldus(1)

- C lubab programmil *heap*ist jooksvalt mälu võtta. Seda piirab vaid jooksmise ajal saadaval oleva mälu hulk.
- Selleks on 3 (stdlib.h all defineeritud) funktsiooni:

```
void *malloc(mitu_baiti_anda);
void *calloc(mitu_on, mis_suurusega);
void *realloc(pointer, mitu_baiti_anda);
```

- Kui mälu hõivamine ei õnnestu, tagastub NULL
- Kuna tagastatakse void tüüpi pointer, tehakse *cast*.

```
int *pi = (int*)malloc(5*sizeof(int)); /* või */
int *pi = (int*)calloc(5, sizeof(int));
pi = (int*)realloc(pi, 10*sizeof(int));
```

Dünaamiline mäluhaldus(2)

- Millal seda tarvis läheb?
  - Näiteks, kui massiivi suurus antakse programmile argumendina
  - Üldreegel: Alati, kui programm kompileerimise ajal ei tea kui palju mingi asi ruumi võtab
- OLULINE: Pärast võetud mälu kasutamist tuleb see süsteemile tagasi anda:

```
void free(void*);
```

- Kui seda ei tehta, tekib mäluleke: kasutamata mälu tundub süsteemile "lukus" ja seda ei saa kasutada.

Mäluhalduse näide

- Korrektne näide

```
int *vec;
if ((vec=(int*)malloc(ARR_LNG*sizeof(int)))==NULL) {
   fprintf(stderr, "cannot allocate\n");
   exit(1);
}
if ((vec = (int*)realloc(vec, NEW_ARR_LNG*sizeof(int)))
   == NULL) {
   fprintf(stderr, "cannot allocate\n");
   exit(1);
}
```

- Üks võimalik viga: ripakil pointer (*dangling pointer*), tekib siis, kui pointer viitab juba vabastatud kohta.

Mäluhalduse halb näide

- Vea näide:

```
int *vec, *new_vec;
if ((vec=(int*)malloc(ARR_LNG*sizeof(int)))==NULL) {
  fprintf(stderr, "cannot allocate\n");
  exit(1);
}
if ((new_vec =
    (int*)realloc(vec, NEW_ARR_LNG*sizeof(int)))
      == NULL) {
  fprintf(stderr, "cannot allocate\n");
  exit(1);
}
```

- vec näitab nüüd kuhu juhtub

Veel üks ripakil pointer

```
char *foo(char *s) {
   char buf[100];
   strncpy(buf, s, 99);
   return buf;
}

main() {
   char *t;
   t=foo(“Hello”); // t on ripakil
}
```

Pointerid funktsioonidele

- Võib tekkida situatsioon, kus tuleb välja kutsuda funktsioon, kuid pole teada milline

```
void *v1, *v2;
if (compare (v1, v2) == 1) { ...
```

v1 võib viidata stringile või integerile. Tuleks kutsuda välja objektitüübile vastav funktsioon...

```
enum type {INT, STR};
int (*compare)(void*, void*); /* pointer funktsioonile */
...
switch (type) {
   case INT: compare = &num_compare; break;
   case STR: compare = &strcmp; break; }
if ((*compare)(v1,v2) == 0) { ... /* või "!
   compare(v1,v2)"*/
```

Pointerid funktsioonidele

- Teine situatsioon, kus vajame pointerit funktsioonile on see, kui kasutame funktsiooni argumendina teisele funktsioonile

```
void string_manipulation(char *s, int (*chr_mnp)(int))
{
  while ( *s !='\0') {
   *s = chr_mnp(*s);
   s++;
  }
}
/* Use of that function * /
char str[10] = "aBcD";
...
string_manipulation(str, tolower);
```

Viimane slaid

- Alustage nihelemisega
- Lobisege ja lahkuge